

The Utility and Practicality of Quantifying Software Reliability

Rob Ashmore

Defence Science and Technology Laboratory
Portsdown Hill Road, Fareham
Hants. PO17 6AD
Email: rdashmore@dstl.gov.uk

Abstract—We argue that quantifying software reliability is important in demonstrating that system-level risks are As Low As Reasonably Practicable (ALARP). Furthermore, we demonstrate that such quantification is possible in at least one meaningful case. It is, however, unlikely to be practical in every case. This means it is unlikely to be included as an explicit objective in standards. Hence, for those cases where software reliability can be quantified, merely following a standard may lead to risk-reduction opportunities being missed.

I. INTRODUCTION

One way of assessing the efficacy of standards is demonstrating the efficacy of part, or parts, of a standard. We are primarily interested in the design, implementation and testing phases of software development. More specifically, we wish to quantify the probability that software will act as specified in its requirements.

II. QUALITATIVE APPROACHES

The efficacy of qualitative (*e.g.* process-based) approaches to software development is demonstrated by the large volume of safety-critical software in use today. However, these approaches do not allow software reliability to be quantified in system fault trees. Whilst this lack of quantification is a common approach, and is consistent with Aerospace Recommended Practice (ARP) 4761 [1], it is not entirely benign.

Consider the simple fault tree shown in Figure 1. We assume that element A is implemented in software and that B and C are implemented via physical means. In this arrangement software and a mechanical item provide mutual back-up.

Suppose that the failure rates of B and C are 10^{-2} and 10^{-5} , respectively. One common approach, involves giving A, the software component, an implausibly low failure rate (*e.g.* 10^{-31}). This rate dominates the AND gate at the bottom left. As a result, the dominant factor into the top OR gate is C's failure rate. Any attempts to reduce the overall failure rate will begin by focussing on C.

Now, suppose that we were only able to justify a failure rate of 10^{-2} for the software component (*i.e.* for A). In this case the top OR gate is dominated by the combination of A and B. Hence, improvements to the overall failure rate will initially focus on these components.

This example is obviously highly simplified: the fault tree only includes three components and, more significantly, it

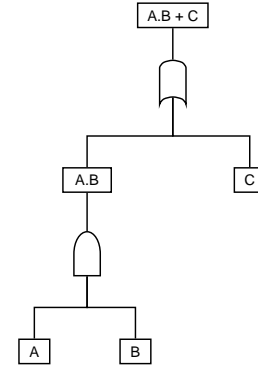


Fig. 1. Example Fault Tree

is highly unlikely that decisions on which components to “improve” will be based solely on mathematical calculations. However, the underlying point remains valid, namely that assuming extremely high rates of software reliability can highlight the *wrong* system components¹. This can lead to scarce resources, both intellectual and physical (*e.g.* power and weight budgets), being used in a sub-optimal manner, potentially jeopardising claims that risks are ALARP.

III. QUANTIFICATION THEORIES

The quantification of software reliability is not new: quantified estimates of the Sizewell B Plant Protection Software (PPS) reliability were produced in the 1990s [2]. Despite this history, it is not a widely used technique. This may be partly explained by philosophical concerns (*e.g.* the fact that software does not “wear out” like physical components do), which are outside the scope of this paper. Until recently, there have also been practical barriers to demonstrating meaningful levels of software reliability for non-trivial cases.

One simple approach to quantifying software reliability uses a “balls and urn” model, where individual balls represent specific inputs and the distribution of balls represents the distribution of inputs the software is expected to see in

¹For reasons of brevity we do not fully detail the argument here, but a similar argument shows that related problems arise if software is given an extremely high failure rate, which is another commonly adopted approach.

operational use. Subject to certain assumptions, successfully completing 4.61×10^6 tests is sufficient to claim a reliability of 1×10^{-6} with 99% confidence [3].

Our point is not that great faith should be put in the precise accuracy of these numbers; like all values used in fault trees there will inevitably be some uncertainty and inaccuracy. However, simply being able to estimate software reliability to within, say, an order of magnitude will allow meaningful system-level decisions to be made.

Other approaches to quantifying software reliability may also be feasible. Consider a piece of monitoring software that initiates “make safe” actions under certain conditions. If the number of different inputs that meet these conditions is suitably small (*e.g.* millions) then all of them may be explicitly tested. This would allow a numerical value to be assigned to a part of a fault tree that considered something like: “software fails to request make safe when conditions require it”. Of course, any safety argument that used this “partial exhaustive” testing would also need to consider the implications of “make safe” being enacted when it was not required.

In a small number of cases it may be possible to test the full input domain of the software. We have recently implemented an automated test environment, which was used to analyse (amongst other things) a currently-fielded control algorithm. Even a naive implementation of the test environment, running on commodity hardware, achieved 320 test cases per second per core². This suggests that within 24 hours a modest 32-core system would be capable of exhaustively testing software that takes a combination of three 8-bit integers and three 1-bit flags.

Our experiences show that exhaustive testing of meaningful algorithms is now within reach. Of course, it is trivially easy to produce algorithms that cannot be exhaustively tested. However, we conjecture that, even if exhaustive testing is impossible, many examples of embedded software are sufficiently small to allow millions of tests to be completed easily. This opens up the possibility of providing evidence to support statistical claims regarding software reliability.

IV. THREATS TO VALIDITY

Our focus on software testing does not cover the case where the original requirements may be incorrect. Likewise, we have ignored interactions between software and people.

We have also implicitly assumed that the test hardware faithfully replicates the operational hardware, which is one of the assumptions underpinning the relationship between the number of successful tests and a quantified reliability claim: recent work on emulation technologies may help demonstrate that the required faithful replication has been achieved.

Another assumption underpinning the relationship between estimated reliability and the number of successfully completed tests is that the distribution of test cases accurately reflects the software’s operational input distribution. Estimates of this

distribution may be informed by: discussions with stakeholders [4]; development activities; and in-service data. Several different distributions could even be used. Whilst this would increase the number of tests it would not reach an impractical level.

Automatically conducting large numbers of tests needs an automated way of checking the result: that is, we need a test oracle. In some cases this may be easy to produce (*e.g.* checking a “make safe” flag, or animating a formal specification). In others it may be more difficult to capture the *entire* set of requirements in an oracle. However, even in these cases testing could confirm, for example, that function pre- and post-conditions are highly likely to be satisfied. This could inform key parts of system fault trees.

There are types of software that by their very nature pose particular problems to quantifying reliability. The lack of knowledge of internal state information means “black box” software is one example; real-time software is another [5]. Adapting the “balls and urn” model indicates that, for example, 4.61 million hours of testing would be required. (The move from “number of tests” to “hours of testing” reflects the desire to talk about “probability of failure per hour” rather than “probability of failure per demand”.) Testing for millions of hours is stretching the bounds of plausibility. That said, if the software is performing a monitoring function then, as discussed earlier, it may be possible to do a partial exhaustive test on those input sets that require action.

V. CONCLUSION

The approaches to quantifying software reliability discussed above will not be suitable for every piece of software. This means they are highly unlikely to be mandated in a standard. However, when they are suitable these methods provide valuable support to system-level risk management. Without this support risk-reduction opportunities may be missed, potentially jeopardising claims that risks are ALARP.

Disclaimer: This article is an overview of MOD sponsored research and is released for informational purposes only. The contents of this article should not be interpreted as representing the views of the MOD, nor should it be assumed that they reflect any current or future MOD policy. The information contained in this article cannot supersede any statutory or contractual requirements or liabilities and is offered without prejudice or commitment.

©British Crown copyright 2014 Dstl. Published with the permission of the Controller of Her Majesty’s Stationery Office

REFERENCES

- [1] SAE International, *ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, Std., December 1996.
- [2] J. May, G. Hughes, and A. D. Lunn, “Reliability estimation from appropriate testing of plant protection software,” *Software Engineering Journal*, vol. 10, no. 6, pp. 206 – 218, November 1995.
- [3] P. Bishop, R. Bloomfield, B. Littlewood, A. Povyakalo, and D. Wright, “Toward a formalism for conservative claims about the dependability of software-based systems,” *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 708 – 717, sept.-oct. 2011.
- [4] J. Musa, “Operational profiles in software-reliability engineering,” *Software, IEEE*, vol. 10, no. 2, pp. 14 – 32, March 1993.
- [5] R. Butler and G. Finelli, “The infeasibility of quantifying the reliability of life-critical real-time software,” *Software Engineering, IEEE Transactions on*, vol. 19, no. 1, pp. 3 – 12, January 1993.

²In this case most time is spent running the test oracle, which is an animated Vienna Development Method (VDM) specification. Running just the Software Under Test (SUT) we can achieve around a million cases per second per core.